Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 116 (2005) 113–131

www.elsevier.com/locate/entcs

QUACK: A Platform for the Quality of New Generation Integrated Embedded Systems [1]

Mauro Pezzè [2] Andrea Baldini [3] Giovanni Denaro [2]
Giuseppe Lipari [4] Matteo Rossi [5] Davide Rogai [6]

Abstract

Over the last two years, the QUACK project investigated a new methodology for assessing the quality of heterogeneous, modular and configurable embedded systems, i.e., systems made out of a number of hardware and software components, usually embedded in devices with real-time requirements, and produced in families of different versions and configurations. The main aim of the project was to overcome the limitations of traditional techniques in dealing with the many new aspects and issues that arise, when the addressed class of systems is under concern. This paper surveys the final results of the QUACK project: that is, a new methodology for quality assessment of heterogeneous, modular and configurable embedded systems, throughout the whole software process.

Keywords: embedded system, quality assessment, heterogeneity, modularity

[1] This work has been partially supported by the Italian Ministry of University and Research within the COFIN 2001 project "Quack: a platform for the quality of new generation integrated embedded systems"

[2] G. Denaro and M. Pezzè are with Università degli Studi di Milano Bicocca, via Bicocca degli Arcimboldi 8, I-20126, Milano, Italy. Email: {denaro, pezze}@disco.unimib.it

[3] A. Baldini is with Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129, Torino, Italy. Email: baldini@polito.it

[4] G. Lipari is with Scuola Superiore S. Anna, Piazza Martiri della Libertà 33, I-56127, Pisa, Italy. Email: lipari@sssup.it

[5] M. Rossi is with Politecnico di Milano, Via Ponzio 34/5, I-20133, Milano, Italy. Email: rossi@elet.polimi.it

[6] D. Rogai is with Università degli Studi di Firenze, Via di Santa Marta 3, I-50139, Firenze, Italy. Email: rogai@dsi.unifi.it

doi:10.1016/j.entcs.2004.02.082

1 Introduction

Heterogeneous, modular and configurable embedded systems are composed of several heterogeneous components that share hardware and software resources. The systems of this class are generally available in several versions and configurations that are obtained by suitably combining different subsets of components. The components can be in turn characterized by different real-time and safety requirements even within the same system. This class of systems is currently applied in several application domains, which include, but are not limited to, automotive, railways, space, defense, wearable computing, and domotic.

The board systems of new generation cars well exemplify this category of systems. They combine several hardware devices, e.g., DVD, GSM and GPS devices, and provide many complex software services, e.g., Internet facilities, car alarm monitoring and integrated control of all available devices. Components have safety and real-time requirements at different criticality levels: for example, a failure of the car alarm monitoring system may have severe consequences, while a failure of the Internet facilities does not produce major damages. Different car models use various configurations of the available components, for example, base models may not include GPS and Internet facilities, while top models are likely to include all facilities. Configurations are obtained by changing, adding or updating subsets of components.

The critical nature of these systems entails high quality requirements that cannot be easily satisfied by means of traditional test and analysis techniques, which are not able to master the complexity of these systems and do not effectively deal with many variants of the same system. It is often impractical to pursue the independent verification of the many configurations and versions in which such systems are available. The development of increasingly many varieties of configurations and versions of systems with high quality requirements, demands new techniques that allow to reuse knowledge on both the quality of components and their similarities among different configurations, such to reduce the checks on each system.

The QUACK project studied over the last two years a new methodology to control the quality of heterogeneous, modular and configurable embedded systems. The proposed methodology (that we call *QUACK methodology* after the project acronym) merges different techniques to be used at various development phases or for different types of requirements. The QUACK methodology includes formal methods to check initial requirements and to derive system tests; run time monitoring techniques to automatically capture the behavior of components in different configurations and compare these behaviors across the configurations, aiming at identifying potential inconsistencies; schedulability

analysis and model checking to verify properties of systems with real-time and fault-tolerance requirements; test case generation for embedded components and automatic deployment of the test cases in the test environment.

This paper reports the results of the QUACK project. Section 2 overviews the QUACK methodology, indicating the proposed techniques as well as the addressed phases and requirements. Section 3 presents the methods proposed for formalizing system requirements, checking their validity and generating system tests. Section 4 presents the technology proposed for generating quality obligations and test cases for components and component-based software systems. Section 5 presents the methodologies proposed for verifying real-time and fault-tolerance requirements. Section 6 presents the technique proposed for automatically deriving test cases from UML design documents. Finally, Section 7 summarizes the results of the project and outlines ongoing research work.

2 An Integrated Approach to the Quality of Heterogeneous, Modular and Configurable Embedded Systems

Figure 1 illustrates the main elements of the QUACK methodology and indicates the sections of this paper where the corresponding approaches are presented. The figure identifies the needs of support for quality assessment at three main phases of the development of heterogeneous, modular and configurable embedded systems.

Specification: the development of a new version or configuration of a system, starts analyzing the user requirements for the new system and defining the system specification, i.e., a document that identifies the expected behavior independently from the internal architecture of the system. In the current practice, systems specifications are often expressed in natural language.

Design: system specifications are mapped onto a suitable architecture based on the examination of existing versions and configurations (System Family) and identifying the components to be reused, adapted, added or substituted to turn an existing configuration in a new one. Modular design is often expressed in diagrammatic visual languages, e.g., suitable subsets of UML.

Deployment: the final components, either new or reused, are integrated in the software architecture to produce the new system. Suitable middleware technologies, e.g., the Java 2 Enterprise Edition platform [46], can facilitate the deployment.

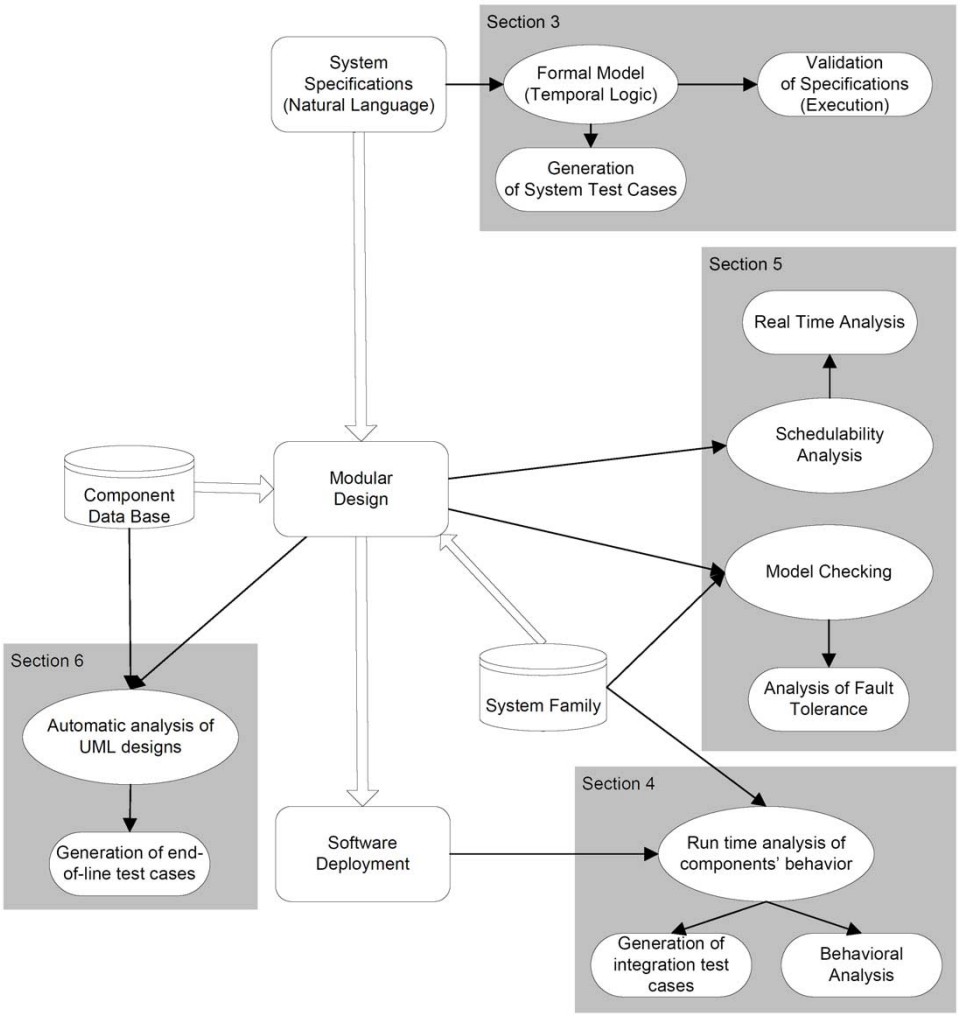

Fig. 1. The main elements of the QUACK methodology

The QUACK methodology includes four sets of methods for supporting quality assessment of systems at each phase, thus aiming at anticipating as much as possible fault identification and removal. Each set of methods takes advantage from the different information that are available at the specific phase. According to the QUACK methodology:

(i) System specifications are formalized by means of temporal logic. The formal specifications are executed to identify inconsistencies and incompleteness of the specified requirements, and used to automatically generate test suites for system testing.

(ii) Run-time monitors collect data on both the functional behavior of the different components and the dynamic interaction patterns within existing configurations and versions. The collected data are used to generate invariants that support the identification of potential faulty behaviors for new versions and configurations. Integration test suites are generated based on the invariants.

(iii) Schedulability analysis and model checking are used to verify real-time and fault-tolerance properties addressing the component-based nature of the systems.

(iv) Automatic analysis of design diagrams expressed in UML, supports the automatic generation of test suites for end-of-production testing of both the final systems and their components.

The proposed techniques are highly automated, thus contributing to improve the quality of the final system with a low impact on the overall costs of the development process. The integration of the different methods is very light: although the best results are achieved with the integrated methodology, each method can be independently used to verify systems that do not require all analysis steps.

3 Formal Specification of Real Time Embedded Systems

Initial (informal) specifications are translated into temporal logic specification for early analysis that includes:

- Component-based verification of systems through model checking.
- Deductive verification of modular systems thorough theorem proving.
- Specification of real-time systems through states and events.
- Code derivation of real-time systems by direct execution of their behavior specification.

This section describes how the above approaches have been implemented in the QUACK methodology.

As the range of issues covered by the QUACK methodology is wide, different, complementary formalisms have been developed and studied. The approaches explored are based on three temporal logics: OTL [24], which is compatible with UML; TRIO [20], which is suitable for specification and verification (through model checking and theorem proving) of both discrete and continuous real-time systems; TILCO [31], which can produce expressive,

directly executable and verifiable specifications, from which it is possible to automatically generate code.

Despite its wide acceptance in industry, UML still lacks powerful modeling mechanisms for effectively dealing with temporal constraints. Lavazza, Morasca and Morzenti presented an extension to the UML Object Constraint Language (OCL) [36], called Object Temporal Logic (OTL) [24] to deal with timing aspects.

OTL simply adds two classes, `Time` and `Offset` to the OCL 2.0 standard library. Class `Time` models time instants, while class `Offset` models the distance between two time instants. Time instants are referred to the current time, which plays the role of time origin.

Classes `Time` and `Offset` allow to define the typical temporal operators of temporal logics (i.e. *Always*, *Sometimes*, *Until*, etc.), and thus model timing aspects. OTL allows users to reason about time in a quantitative fashion, so that it is possible to express properties like *event B must occur at most l time units after event A*. `Time` and `Offset` may be *discrete* or *dense*, depending on the modeled application, thus allowing users great flexibility and expressive power.

OTL formulas are evaluated with respect to the current time instant, which is left implicit. Primitive `eval` of class `Time` is used in OCL to evaluate predicates at time instants different from the current one. For example, given an object `t` of class `Time` and a predicate `p` (which is an `OclExpression`), `t.eval(p)` returns true (a boolean value) if `p` holds at time instant `t`.

Further details about the use of OTL in the QUACK methodology can be found in this volume in the paper by Lavazza et al. [25].

While OTL is a very promising approach to the problem of modeling the behavior of object-oriented applications, it is quite young; in addition, a set of tools for the *verification* of systems specified with OTL is currently lacking. Another, more mature, logic-based approach to modeling real time systems is the TRIO temporal logic [20,35]. To support the QUACK methodology, tools and techniques for the (semi)automatic formal verification of modular real time systems specified with TRIO have been developed.

The quack methodology supports the verification of applications though both model checking and theorem proving. While many current model checkers (and most notably the popular SPIN [21]) use Linear Temporal Logic (LTL, [41]), which only has future temporal operators, to express properties of transition systems, TRIO [20] can express properties both in the future and in the past with respect to the current instant (which is left implicit in TRIO formulae). The QUACK project studied how formulae with both past and future operators can be treated in SPIN (see [42] for details), and how to apply

those principles to perform model checking with TRIO formulae (see [34] for details).

The technique described in [34] is centered around the concept of an "event generator" that explores the space of the input variables for the different modules composing the system. An exhaustive search of that space would produce a combinatorial explosion in the complexity of the algorithm, so some optimizations are introduced in order for the event generator-based approach to be viable. These optimizations focus around the concept of "input" and "output" variables for a TRIO module and, in particular, on the idea that an "output" variable can be *computed* from the inputs, instead of simply generated.

The QUACK methodology complements model checking with theorem proving of TRIO specification focusing on the modular characteristics of the language that allow to specify component-based systems. To support TRIO-based theorem proving in the QUACK methodology, the encoding of TRIO in PVS has been revised and extended, and a compositional framework for PVS-supported deductive proofs with TRIO has been developed.

The compositional framework is centered around a new temporal operator ($\stackrel{+}{\rightarrow}$, called "while-plus"), and on a sound compositional inference rule based on it.

Here we outline the approach with a simple example. Let S be a system composed by n modules $C_1, \ldots, C_n$. Let each module $i = 1, \ldots, n$ be associated with an assumption E_i about the behavior of its environment and a behavioral property M_i of the module itself, i.e., each module $i = 1, \ldots, n$ is associated with a specification of the form: *assuming* the environment of C_i behaves as in E_i, we can *guarantee* that the module behaves as in M_i. In general, the system S has its own environment it interacts with. Let E be the assumption we make on S's environment and M the global property we want to prove of S. S is characterized by a *global* specification of the form: assuming the environment of S behaves as in E, we can guarantee that the composite module behaves as in M.

The compositional inference rule developed for the TRIO language lets us derive the validity of the global specification of S from the validity of the local specifications of the C_is.

Further details about TRIO-based theorem proving in the QUACK technology can be found in this volume in the paper by Furia and Rossi [17].

A notion of compositionality in the context of theorem proving has been explored also for TILCO. The TILCO temporal logic uses theorem proving to ensure important properties of a TILCO-written specification, such as safety

and liveness. TILCO expressive operators (dynamic interval, bounded happen) showed in [8] allow to express very complex properties in a readable form. The extension of TILCO presented in [7] (CTILCO, i.e., Communicating TILCO) provides a new model for composition/decomposition of complex systems and for process communication. In CTILCO a specification can be obtained from several interconnected processes, and theorem proving can be performed to validate the correctness of the interaction between different processes. In the context of the QUACK methodology the use of PVS has been studied, and strategies to prove TILCO expressions have been produced; such strategies are based on ad-hoc lemmas which are tailored on the basic TILCO operators.

The benefits of modeling reactive, embedded time-critical systems with a *combination* of state-like and event-like concepts (as opposed to restricting the specification alphabet to either kind of item) is discussed in [19].

A case study about the application of CTILCO according as part of the QUACK methodology can be found in this volume in the paper by Bellini et al [9].

The most important feature of the TILCO approach is the executability of specifications [5]. A TILCO specification can be executed in a real-time context, where the computational cost to determine the system outputs at each instant is bounded and predictable. The execution process requires a transformation of the TILCO rules in a lower-level language called Basic Temporal Logic. Thanks to this formalism it is possible to obtain a Temporal Inference Network, the evolution of which generates the system reactive behavior. This process has been automated by a suitable compiler to produce a file which models the inference network; this file is loaded by the TILCO Executor, which produces the outputs on the basis of input histories.

In the QUACK methodology, this development step to implement real-time systems from specification execution has been improved and extended. A new integrated development environment, called Dev-TILCO, has been created to better answer to the industrial request of a high-productive development tool. In Dev-TILCO a formal specification design can be performed by taking into account formal parts and traditional programming language modules (C++ sources) [6]. The development process starts from a design which separates the temporal behavior of single processes and interactions from the executable code. The names of the temporal predicates (input or output) related to system events are declared and used in the specification. These declarations are automatically processed to obtain programming language objects, which are connected to the TILCO executor (to execute the behavior specification) and are usable together with the other sources of the application, in order to build

an executable program ruled by a formal description of the time constraints. Like in event-driven programming predicates events are connected to special functions which are customizable to interact with the "classic environment" of the program (like typed internal variables, functions or threads).

4 Test of New Configurations of Component-Based Systems

The development of a new heterogeneous, modular and configurable embedded system is often based on modifying a subset of modules of a system of the same family already in use. In the general setting, some components are eliminated, others are modified, and yet others are added. Very frequently already existing components are reused, directly or with small modifications.

The integration of new components can result in failures due to subtle differences in the interactions triggered by the new components. For example, let us consider a system in which a component is responsible for dynamically selecting the best catalog to locate certain items. Furthermore, let us consider that this component substitutes and updates a previous component that, in another configuration of the same system, was able to locate items in a fixed catalog. In this case, the new system may fail when catalogs cannot be dynamically located or when they become unavailable under some execution conditions, which was never the case in the previous configuration. The new system may behave correctly in many cases and fail only in particular situations.

The issue of testing and analyzing component-based software has been studied in the last year [44,40], but the solutions proposed so far do not address all aspects of the problems. Design-for-testability techniques proposed by Binder and extended by the self-test components proposed by Martins et al., assume that components are suitably instrumented with built in facilities for testing [11,30]. Retrocomponents proposed by Liu and Richardson require significant work of tester designers and thus imply substantial costs [28]. BIT wrappers proposed by Edwards require detailed knowledge of the design of components ad thus do not always apply to third parties COTS [12].

The technique developed during the QUACK project springs from the ideas underlying perpetual testing of Pavlopoulou and Young [39] and takes advantage from the technique for invariant detection by Ernst et al. [13]. The technique requires no specific knowledge of the internal structure of components nor detailed understanding of specifications, thus overcoming the main usability limitations of previous work.

The technique applies to all cases in which one or more components are

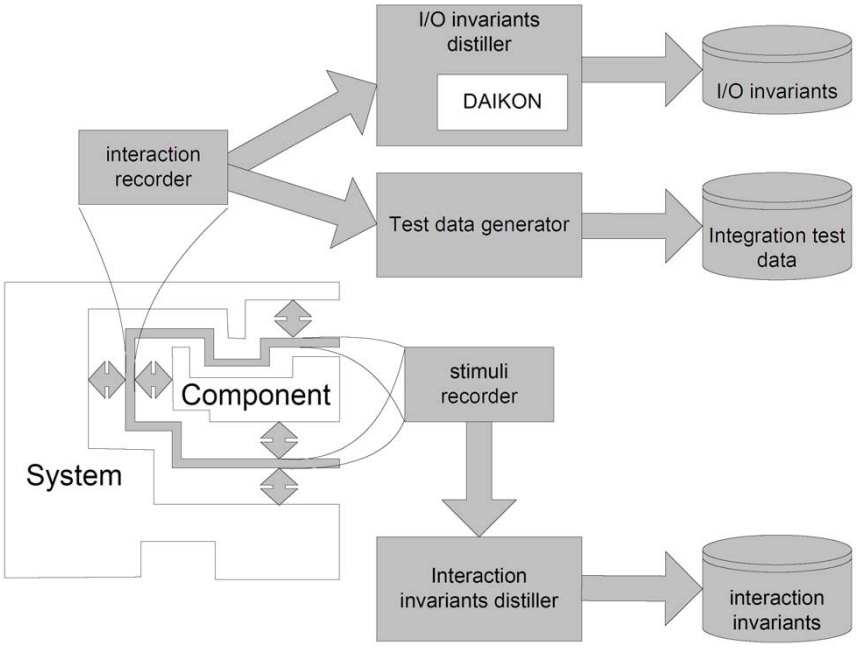

Fig. 2. Automatic recording of components' behaviors. Interaction and stimuli recorders are inserted between the system and the monitored component.

substituted with new ones for obtaining a new system, and is based on three main phases: *data gathering*, *invariant detection*, and *component verification*. In the data gathering phase, the target systems are automatically instrumented to collect information about the interactions between the components and between the components and the environment. The invariant detection phase distills the information collected at runtime into:

I/O invariants that describe properties of the data exchanged between the system and the monitored component,

interaction invariants that describe properties of the interaction patterns of the monitored component with the other components of the system,

integration test data that represent the recorder interactions.

The recording process is illustrated in Figure 2. Interaction and stimuli recorders capture the interactions of the components with the systems. They can be based on different technology. In the QUACK prototype, we designed non-intrusive recorders for Java that are bound at system start-up. The I/O invariants distiller "flattens" the objects exchanged between the component and the system by recursively extracting the key information up to a given depth, using automatically identified "inspectors". I/O invariants are

extracted from the flattened data by means of Daikon, a tool by Ernst et al. [13], which applies to simple scalar variables and collections. The Interaction Invariant distiller computes a regular expression that summarizes the interaction patterns between the component and the system. Finally, the Test data generator filters the recorder data according to different criteria to select a reasonable subset of the collected data that is suitable for integration testing.

In the component verification phase, new components are verified and tested for compatibility with the system. Verification is based on the invariants collected in the data gathering phase. When substituting a component with a new one already in use in other versions or configurations, the invariants of the new component are checked against the system invariants. The identified differences are then analyzed to discover potential problems in the new system. Testing is based on the execution of the integration test data collected in the data gathering phase. Oracles are automatically derived from system invariants.

Further details about the QUACK technology for testing new configuration of component based systems can be found in this volume in the paper by Mariani and Pezzè [29].

5 Verification of non functional properties

The correctness of a real-time system depends not only on the correctness of the produced results but also on the time at which they are produced. Temporal logic specifications support the analysis of temporal properties of early requirements. Temporal constraints must be verified also at design and deployment level. At design level, we assign a a worst case execution time to each activity and we compute the worst case interleaving of all the activities to see if the constraints are satisfied.

Most of the research on component based real-time embedded system is related to the software design phase. Only recently non-functional constraints like deadline are being taken into consideration. In particular, component based software development techniques have been only recently applied to the design and implementation of safety critical real-time systems with little practical results. Many problems should be addressed before being able to successfully apply component based design methodologies in real-time systems.

Real-time systems often consist of several concurrent cyclic tasks. Therefore, the first requirement for their components is being multi-threaded, i.e., components must be able to execute a set of concurrent threads. No methodology addresses explicitly the design of a component containing more than

one thread. The OMG has proposed the UML-RT profile for schedulability, performance and time specification [37]. This profile allows the design of real-time applications with UML. However, the profile is not well suited for component based design. Isovic, Lindgren and Crnkovic [23] presented a similar idea in the context of the slot shifting scheduler [16], but in their approach components consist of one single thread.

The QUACK methodology addresses the problem of specifying components containing sets of execution threads, focusing in particular on the problem of specifying the scheduling strategy for each component. In fact, in a multi-threaded real-time system, the scheduler plays an important role. Fixed priority scheduling is commonly used in real-time systems, where each thread is assigned a fixed priority expressed an integer number. However, when a component is developed in isolation, it is not clear how priorities should be assigned. The developers of one component can fix the priorities of the threads inside the component as relative priorities, but they have no idea of the absolute values that these priorities will have in the final system. Moreover, different components may want to use different scheduling strategies, such as, Earliest Deadline First (EDF), static or non-preemptive scheduling. If different components require different schedulers, than it becomes impossible to assemble all components in the final system.

The approach proposed in QUACK uses a hierarchical scheduling framework: each component can specify its own scheduler; The system implements a global scheduler that selects the components to be executed, and invokes the schedulers of the selected components to choose the thread to be executed.

Quack considers also the use of temporal isolation techniques: Since each component is developed in isolation, it is necessary to analyze and test it in isolation before the final system assembly. Quack proposes a technique to "isolate" the temporal behavior of each component from the misbehaviors of the others.

A general methodology for temporal protection in real-time system is the resource reservation framework [32,33]. The basic idea, which was formalized by Rajkumar [43], is that each task is assigned a *server* that is reserved a fraction of the processor available bandwidth: if the task tries to use more than it has been assigned, it is *slowed down*.

This framework allows a task to execute in a system as if it were executing on a dedicated virtual processor, whose speed is a fraction of the speed of the processor. By using a resource reservation mechanism, the problem of schedulability analysis is reduced to the problem of estimating the computation time of the task without considering the rest of the system.

Recently, many techniques have been proposed for extending the resource

reservation framework to hierarchical scheduling [26,45], but none of them has been done in the context of component based software development so far.

In QUACK, we developed a methodology for specifying and analyzing a component based real-time system, where each component consists of one or more concurrent threads, and can specify its own scheduling strategy. In our approach, each component is assigned a minimal fraction of the processor bandwidth and it is protected from the interference of the other components. We developed a mathematical model of the component to compute the optimal server parameters to be assigned to a component to ensure the satisfaction of the temporal constraints.

Further details on the methodology for specifying and analyzing hierarchical scheduling strategies for component based systems, can be found in this volume in the paper by Lipari et al. [27].

Another non-functional aspect relevant to embedded system is dependability, that is, their ability to behave following predetermined requirements even in the case of a fault in the system itself, or in the external environment.

Fault injection is a technique that can be used to evaluate the dependability of a system, either hardware or software [22], and which involves the study of failures and errors in order to recreate particular failure scenarios to test the fault-tolerant architecture of the system. Bernardeschi, Fantechi and Gnesi proposed a method to inject faults at the specification level, and showed how to use model checking to verify fault tolerance at specification level [10].

QUACK refined and extended the method, and evaluated the results by analyzing the SCA system (Sistema Conta Assi, axes counter system), described in [15]. SCA is a railway signaling device used to decide if a given railway section is free or still occupied by a train. The status of the railway is computed by checking the number of wheelsets entering and leaving specifics PRAs, i.e., detection points on the railway. A PRA is free if the number of the wheelsets entering the section is equal to the ones leaving the section. This information is used by the control and acquisition units, named UCAs, to enable or prevent the next train to enter the section, by means of suitable semaphores.

Injection on the SCA model has been done by connecting a suitable SDL block, which models the Fault Injector, to the SDL model of the UCA unit. This block contains a SDL process, called Fault_Injector_Process, which models the Fault Injection activity with an Extended Finite State Machine (EFSM). This EFSM can be configured accordingly to the type of Fault Injection to be performed.

A suitable SDT simulator interface has been designed with some of the main SCA signals and the commands to configure, activate and stop the In-

jector; the evolution of the system while simulating a particular scenario can be traced using Message Sequence Charts (MSC). Fault-Tolerant behaviors can be verified using one or more MSCs, representing scenarios in which the system has to react in a fault-tolerant way towards wrong signals: if the MSCs is violated, the SDT Validator returns an MSC indicating the way the fault-tolerant scenario has been violated, thus allowing modification of the SDL system to correct the fault.

An elaboration of the model described in [15] is presented in this volume in the paper by Banci et al. [4].

6 End-of-Production Testing

The QUACK methodology includes a technique for end-of-production (EOP) testing for heterogeneous, modular and configurable embedded systems, which is mandatory in industry. This technique enables to translate test scenarios (given by designers or generated from models) to test commands. The test commands are handled by test simulators that surround the system under test. In this way, starting from high-level scenarios, it is possible to automatically generate functional test scenarios using the test environment of common industrial simulators.

This section describes the most critical point of the technique, that is the translation process.

A big gap exists between the design level and the final commands for the test equipments, i.e., between the input and the output of the translation phase. On one hand, in the design phase, the system descriptions are generally provided at a very abstract level. Very often, these descriptions take the form of messages that represent the user view of the system. For example, messages are such as "phone_call (number:123-4567)" and "hang_up" would be very common in the description of a mobile phone. On the other hand, test commands are generally the low-level commands of standard test simulators, stated in the simulator command set and highly customized. For example, typical test commands may refer to a set of test resources, such as, ports and interfaces. To fill in this gap, we define an intermediate concept: the test level messages. The test level messages are the message representation of test commands. For building the test-level messages, we use a bottom up approach, i.e., we start from the command set of the simulators and from the actuators that represent the test environments.

Simulating command messages often requires to set the working environment of the system. For example, simulating the command *CALL* for a cellular phone requires a cellular network to be present in the environment, otherwise

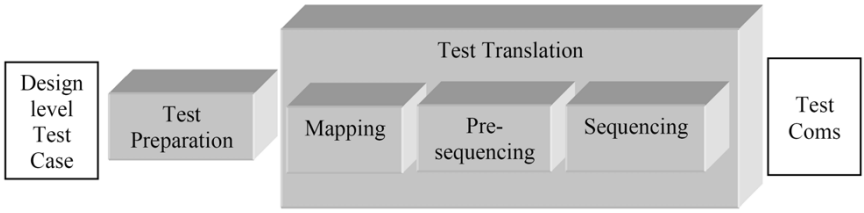

Fig. 3. The translation process

no phone calls can be made. Environment programming is a hard task for test designers because a lot of environment messages are needed and because a wrong setting in the environment may alter the test results. To reduce workload and errors, our technique provides a default environment for each test-level message. The default environments are created by the test engine, along with the default observation of the effects of the messages.

The translation process that set up the testing environment is divided into preparation and translation phases as shown in Figure 3.

The Test Preparation phase ensures that a test case is compliant with the current test environment. This is accomplished by checking the test case against a set of pre-defined requirements (expressed as rules) defined such that, if they are satisfied, the translation is possible. This phase is fully automatic. The Test Translation phase consists of three steps:

the mapping step establishes a correspondence between the information at the design level and the test commands. The input of this step is a design-level test case, previously validated in Test Preparation phase, while the output is a non-empty set of test-level sequence diagrams. The mapping step performs three different actions: it resolves message polymorphism; translates the design layer messages into test layer messages; and adds the default environment and observation messages. A design level message is polymorphic when it can be issued to the system in different ways (for example, using vocal control or the keyboard). This entails that the translation is not unique, i.e., there exist different test messages that correspond to the same design level message. Resolution of polymorphism can be either automatic (following full or partial coverage criteria, e.g., random) or manual (performed by the test designer). Apart from polymorphism, the mapping step is automatic.

the Pre-sequencing step assigns minimum timing attributes to the test messages. The test designer can indicate a specific timing of the messages or assign default timings, with the smallest inter-command delay in order

to keep the test execution time as low as possible.

the sequencing step translates the test-level sequence diagrams generated at the Mapping step along with the timings added by the Pre-sequencing step, into a list of commands directly understandable by the test environment. This step is heavily based on the test engine.

The test engine is a piece of software running on a dedicated workstation; the test engine controls the simulators using time wheels. Time wheels are widely used in the hardware world, in the context of simulation. Delays are tracked using a time wheel divided into ticks or slots, with each slot representing a unit of time. A software pointer marks the current time on the timing wheel. As simulation progresses, the pointer moves forward by one slot for each time step. The event list tracks the events pending and, as the pointer moves, the simulator processes the event list for the current time. On time wheels we directly indicate timed commands to the simulators, so the first operation is a conversion of each test-level message into a set of simulator commands, with existence and uniqueness guaranteed by the Test Preparation phase. The minimum delays between messages inserted during the Pre-sequencing step are inherited by the test commands.

Since each test command has an associated implementation time the test engine must send each command to the appropriate simulator in adequate advance. Moreover, the test engine uses time wheels to track the usage of the test resources. As a matter of fact, even if all timings required to inject the messages are respected, it may happen that an environment setting needed by some further messages is changed before it has been used. In such cases the test is declared failed.

Further details on the translation process can be found in [1,2,3].

7 Conclusions

Heterogeneous, modular and configurable embedded systems are increasingly popular in many application domains because the flexibility introduced with this technology allows for producing complex systems at low cost. However, this class of systems entails new quality requirements that cannot be adequately dealt with traditional specification, testing and analysis techniques.

This paper presented a set of new techniques for assessing the quality of heterogeneous, modular and configurable embedded systems. The presented techniques cover all main quality assessment requirements in the software process for the target class of systems, by supporting: the definition and validation of component-based systems with real-time requirements; the assessment of both isolated components and their integration; the analysis of schedulability

and fault-tolerance properties of a composed system based on the properties of its components; the generation of test cases for embedded components and the automatic deployment of the test cases in the test environment.

So far, we applied the proposed techniques to an initial set of case studies, including industrial systems provided by our research partners, well-known examples taken from the literature in the field, public applications that exemplify the use of component technology. The results are promising, even though preliminary. We are now continuing the experiments, aiming at improving our confidence in and knowledge of the QUACK methodology and at refining and tuning the single techniques.

References

[1] Baldini, A., A. Benso, S. Mo, A. Taddei and P. Prinetto, *A UML process for system level functional test: an industrial perspective*, in: *Proceedings of Integrated Design and Process Tech (IDPT'02), Pasadena, USA*, 2002.

[2] Baldini, A., A. Benso, P. Prinetto, S. Mo and A. Taddei, *Towards a unified test process: from UML to end-of-line functional test*, in: *Proceedings of the IEEE International Test Conference 2001 (ITC'01)*, 2001, pp. 600–608.

[3] Baldini, A., A. Benso, P. Prinetto, S. Mo and A. Taddei, *Efficient design of system test: a layered architecture*, in: *Proceedings of the IEEE International Test Conference 2002 (ITC'02)*, 2002, pp. 930–939.

[4] Banci, M., M. Becucci, A. Fantechi and E. Spinicci, *Validation coverage for a component-based SDL model of a railway signalling system*, in: *Proceedings of the 2nd International Workshop on Test and Analysis of Component-Based Software (TACoS'04)*, 2004.

[5] Bellini, P., A. Giotti and P. Nesi, *Execution of TILCO temporal logic specifications*, in: *Proceedings of the 8th International Conference on Engineering of Complex Computer Systems (ICECCS'02) Greenbelt, Maryland* (2002), pp. 78–88.

[6] Bellini, P., A. Giotti, P. Nesi and D. Rogai, *TILCO temporal logic for real-time systems implementation in C++*, in: *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03) San Francisco Bay* (2003).

[7] Bellini, P. and P. Nesi, *Communicating TILCO: a model for real-time system specification*, in: *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS'01) Skvde, Sweden* (2001), pp. 4–14.

[8] Bellini, P. and P. Nesi, *TILCO-X an extension of tilco temporal logic*, in: *Proceedings of the 7th International Conference on Engineering of Complex Computer Systems (ICECCS'01) Skvde, Sweden* (2001), pp. 15–25.

[9] Bellini, P., P. Nesi and D. Rogai, *Validating component integration with C-TILCO: A case study*, in: *Proceedings of the 2nd International Workshop on Test and Analysis of Component-Based Software (TACoS'04)*, Electronic Notes on Theoretical Computer Science **82(6)**, 2004.

[10] Bernardeschi, C., A. Fantechi and S. Gnesi, *Model checking fault tolerant systems*, Software Testing, Verification and Reliability **12** (2002), pp. 251–275.

[11] Binder, R., *Design for testability in object-oriented systems*, Communications of the ACM **37** (1994), pp. 87–101.

[12] Edwards, S., *A framework for practical, automated black-box testing of component-based software*, Software Testing, Verification and Reliability (STVR) **11** (2001).

[13] Ernst, M., J. Cockrell, W. Griswold and D. Notkin, *Dynamically discovering likely program invariants to support program evolution*, IEEE Transactions on Software Engineering **27** (2001), pp. 99–123.

[14] Fantechi, A. and E. Spinicci, *Fault-injection on SCA SDL model*, Technical report, QUACK Technical Report, DSI (2003).

[15] Fantechi, A. and E. Spinicci, *Modelling and validating a multiple-configuration railway signalling system using SDL*, Electronic Notes in Theoretical Computer Science **82** (2003).

[16] Fohler, G., *Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems.*, in: *Proceedings of the 16th Real Time System Symposium, Pisa, Italy*, 1995.

[17] Furia, C. A. and M. Rossi, *A compositional framework for formally verifying modular systems*, in: *2nd International International Workshop on Test and Analysis of Component Based Systems, Barcelona, Spain*, 2004, submitted for publication.

[18] Gargantini, A. and A. Morzenti, *Automated deductive requirement analysis of critical systems*, ACM Transactions on Software Engineering and Methodology (TOSEM) **10** (2001), pp. 255–307.

[19] Gargantini, A., A. Morzenti and E. Riccobene, *Using counters to model temporal relationships among events*, in: *Proceedings of State-oriented vs. Event-oriented thinking in Requirements Analysis, Formal Specification and Software Engineering, Pisa, Italy*, 2003.

[20] Ghezzi, C., D. Mandrioli and A. Morzenti, *TRIO: A logic language for executable specifications of real-time systems*, The Journal of Systems and Software **12** (1990), pp. 107–123.

[21] Holzmann, G. J., *The SPIN model checker*, IEEE Transactions on Software Engineering **23** (1997), pp. 279–295.

[22] Hsueh, M., T. Tsai and R. Iyer, *Fault injection techniques and tools*, Computer **30** (1997), pp. 75–82.

[23] Isovic, D., M. Lindgren and I. Crnkovic, *System development with real -time components*, in: *Proc. of ECOOP2000 Workshop 22 - Pervasive Component-based systems*, Sophia Antipolis and Cannes, France, 2000.

[24] Lavazza, L., S. Morasca and A. Morzenti, *A dual languae approach to the development of time-critical systems with UML*, in: *Proceedings of the UML'03 workshop on Critical Systems Development with UML, San Francisco, USA*, 2003, report TUM-I0323 of Technische Universitat Munchen.

[25] Lavazza, L., S. Morasca and A. Morzenti, *A dual languae approach to the development of time-critical systems with UML*, in: *Proceedings of the 2nd International Workshop on Test and Analysis of Component-Based Software (TACoS'04)*, Electronic Notes on Theoretical Computer Science **82(6)**, 2004.

[26] Lipari, G. and S. K. Baruah, *A hierarchical extension to the constant bandwidth server framework*, in: *IEEE Proceedings of the 7th Real-Time Systems and Applications Symposium,*, 2001.

[27] Lipari, G., P. Gai, M. Trimarchi, G. Guidi and P. Ancilotti, *A hierarchical framework for component-based real-time systems*, in: *Proceedings of the 2nd International Workshop on Test and Analysis of Component-Based Software (TACoS'04)*, Electronic Notes on Theoretical Computer Science **82(6)**, 2004.

[28] Liu, C. and D. Richardson, *Software components with retrospectors*, in: *Proceedings of the International Workshop on the Role of Software Architecture in Testing and Analysis (ROSATEA)*, 1998.

[29] Mariani, L. and M. Pezzè, *Automatic validation of component-based systems*, in: *Proceedings of the 2nd International Workshop on Test and Analysis of Component-Based Software (TACoS'04)*, Electronic Notes on Theoretical Computer Science **82(6)**, 2004.

[30] Martins, E., C. Toyota and R. Yanagawa, *Constructing self-testable software components*, in: *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)* (2001), pp. 151–160.

[31] Mattolini, R. and P. Nesi, *An interval logic for real-time system specification*, IEEE Transactions on Software Engineering (TSE) **27** (2001), pp. 208–227.

[32] Mercer, C. W., R. Rajkumar and H. Tokuda, *Applying hard real-time technology to multimedia systems*, in: *Workshop on the Role of Real-Time in Multimedia/Interactive Computing System*, 1993.

[33] Mercer, C. W., S. Savage and H. Tokuda, *Processor capacity reserves for multimedia operating systems*, Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg (1993).

[34] Morzenti, A., M. Pradella, P. San Pietro and P. Spoletini, *Model-checking TRIO specifications in SPIN*, in: *Proceedings of 12th International Formal Methods Europe Symposium, Pisa, Italy*, Lecture Notes in Computer Science **2805** (2003).

[35] Morzenti, A. and P. San Pietro, *Object-oriented logical specification of time-critical systems*, ACM TOSEM **3** (1994), pp. 56–98.

[36] Object Management Group, *Response to the UML 2.0 OCL RfP (ad/2000-09-03) revised submission, version 1.6* (2003), OMG Document ad/2003-01-07.

[37] Object Management Group, "UML profile for schedulability, Performance and time," OMG (2003).

[38] Owre, S., J. M. Rushby and N. Shankar, *PVS: A Prototype Verification System*, in: D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction (CADE-11)*, Lecture Notes in Computer Science **607** (1992), pp. 748–752.

[39] Pavlopoulou, C. and M. Young, *Residual test coverage monitoring*, in: *Proceedings of the 21th International Conference on Software Engineering (ICSE'99)* (1999), pp. 277–284.

[40] Pezzè, M., editor, "Proceedings of the International Workshop on Test and Analysis of Component-Based Systems (TACoS'03)," Electronic Notes on Theoretical Comupter Science 82(6), 2003.

[41] Pnueli, A., *The temporal logic of programs*, in: *Proceedings of 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island*, 1977, pp. 46–57.

[42] Pradella, M., P. San Pietro, P. Spoletini and A. Morzenti, *Practical model checking of LTL with past*, in: *Proceedings of 1st International Workshop on Automated Technology for Verification and Analysis, Taiwan*, 2003.

[43] Rajkumar, R., K. Juvva, A. Molano and S. Oikawa, *Resource kernels: A resource-centric approach to real-time and multimedia systems*, in: *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[44] Richardson, D. and P. Inverardi, editors, "ROSATEA: International Workshop on the Role of Software Architecture in Analysis E(and) Testing," 1998.

[45] Saewong, S., R. Rajkumar, J. P. Lehoczky and M. H. Klein, *Analysis of hierarchical fixed-priority scheduling*, in: *Proceedings of the* 14^{th} *IEEE Euromicro Conference on Real-Time Systems*, 2002.

[46] Shannon, B., *Java 2 platform enterprise edition specification, 1.4 - proposed final draft 2*, Technical report, Sun Microsystems, Inc. (2002).